Constraint Satisfaction Problems

Constraint satisfaction problems

What is a CSP?

- Finite set of variables X1, X2, …, Xn
- Finite set of constraints C1, C2, …, Cm
- Non-empty domain of possible values for each variable D_{x1}, D_{x2}, … D_{xn}
- Each constraint Ci limits the values that variables can take, e.g., V1 ≠ V2

Constraint satisfaction problems

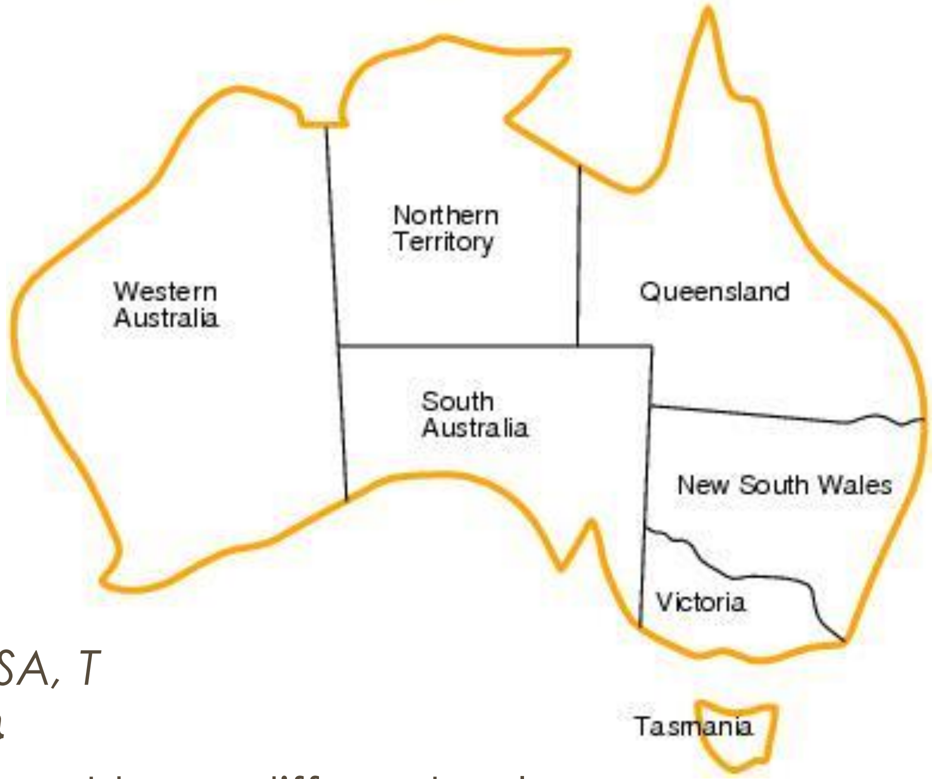
Terminology

- A *state* is defined as an *assignment* of values to some or all variables.
- *Consistent assignment*: assignment does not violate the constraints.
- An assignment is *complete* when every variable is mentioned.
- A *solution* to a CSP is a complete assignment that satisfies all constraints.
    - Some CSPs require a solution that maximizes an *objective function*.

Applications: Scheduling, Planning, Map coloring, etc.

CSP example: map coloring

Example task: colour each region of the map either red, green or blue such that no region is the same colour as any of its neighbours.



CSP Formulation:

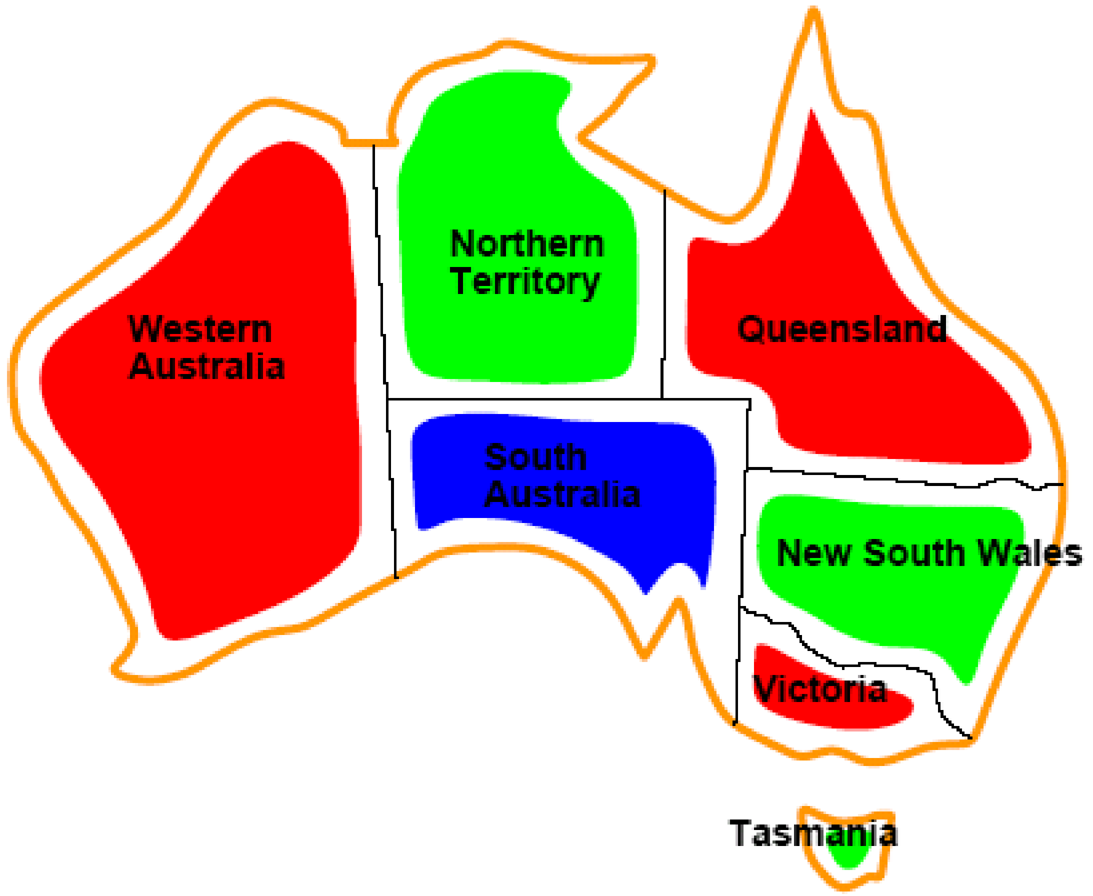
Variables: *WA, NT, Q, NSW, V, SA, T*

Domains: D_i=*{red, green, blue}*

Constraints: adjacent regions must have different colors.

- E.g. *(WA,NT) allowable combinations = {(red,green),(red,blue),(green,red),…}*
- E.g. $WA \neq NT$ (if the language allows this)

CSP example: map coloring



Solutions are assignments satisfying all constraints, e.g.

{WA=red,NT=green,Q=red,NSW=green,V=red,SA=blue,T=green}

Improving backtracking efficiency

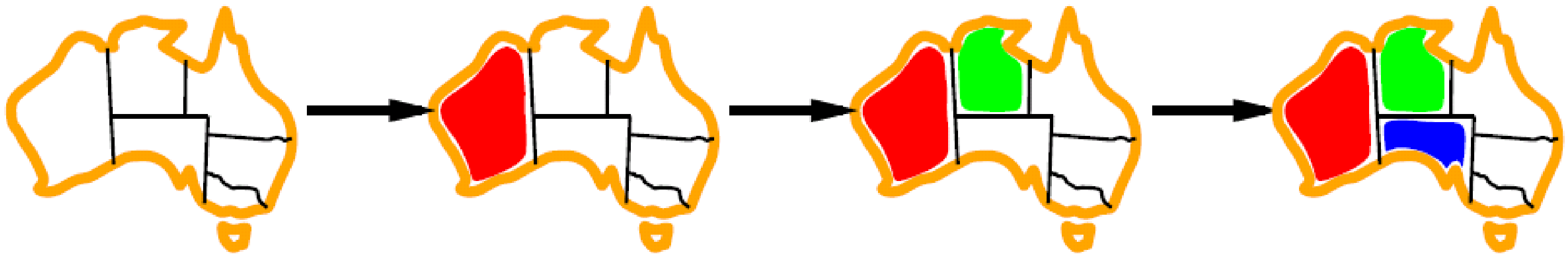
Previously we improved uninformed search algorithms by introducing domain-specific heuristics

We can solve CSPs efficiently without domain-specific knowledge because CSPs conform to a standard representation (set of variables with assigned values)

General-purpose methods that address the following questions can give huge gains in speed:

- Which variable should be assigned next?
- In what order should its values be tried?
- Can we detect inevitable failure early?
- Can we take advantage of problem structure?

Which variable should be assigned next?

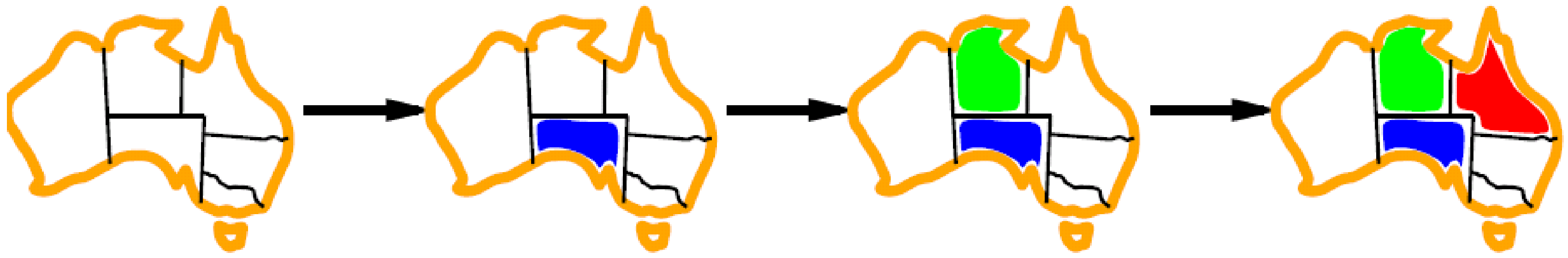


Choose variable with the fewest legal values!

This is called the minimum remaining values (MRV) (a.k.a. most constrained variable, also fail first) heuristic:

- one advantage of this approach is that it picks the variable that is most likely to cause a failure soon, thereby pruning the search tree. If there is a variable X with zero legal values remaining, the MRV heuristic will select X and failure will be detected immediately - avoiding pointless searches through other variables which will always fail when X is finally selected.

Which variable should be assigned next?

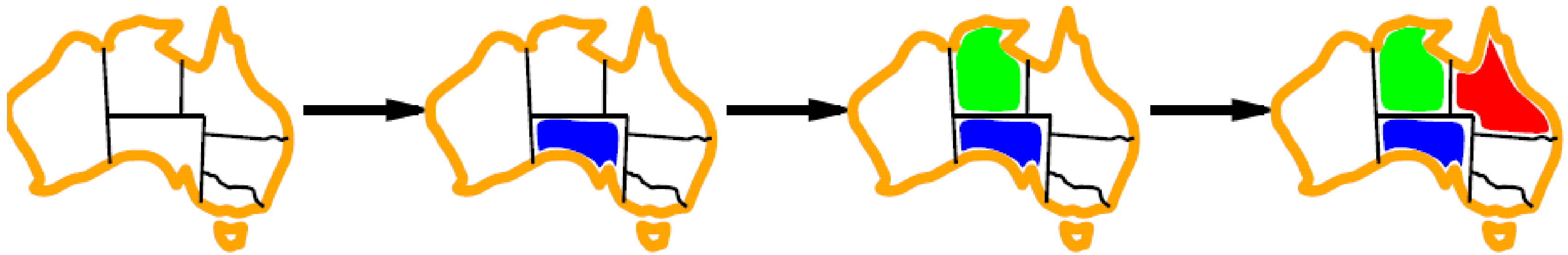


The MRV heuristic doesn't help us with choosing which variable to try *first*?

Degree heuristic: select the variable that is involved in the largest number of constraints on other unassigned variables.

The degree heuristic will hopefully reduce the branching factor on future choices by selecting the variable that is involved in the largest number of constraints on other unassigned variables.

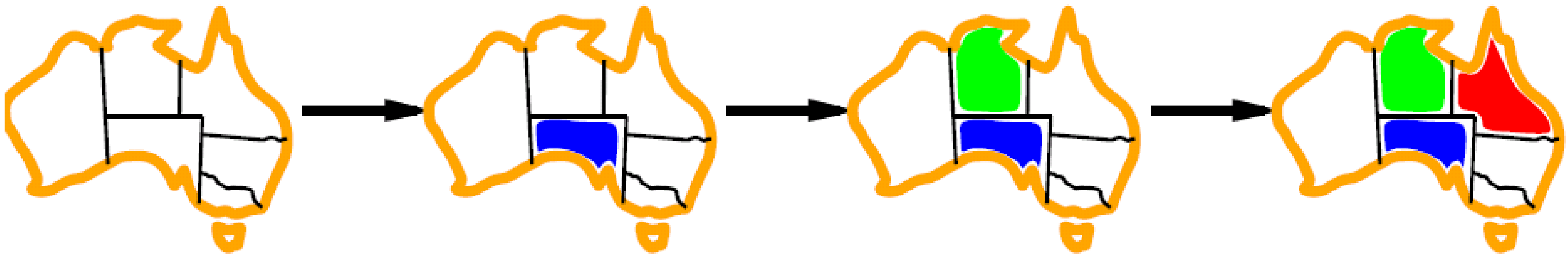
Which variable should be assigned next?



In the Australia map-colouring task SA is the variable with the highest degree (5).

Once SA is chosen, applying the degree heuristic solves the problem without any false steps – you can choose any consistent color at each choice point and still arrive at a solution with no backtracking.

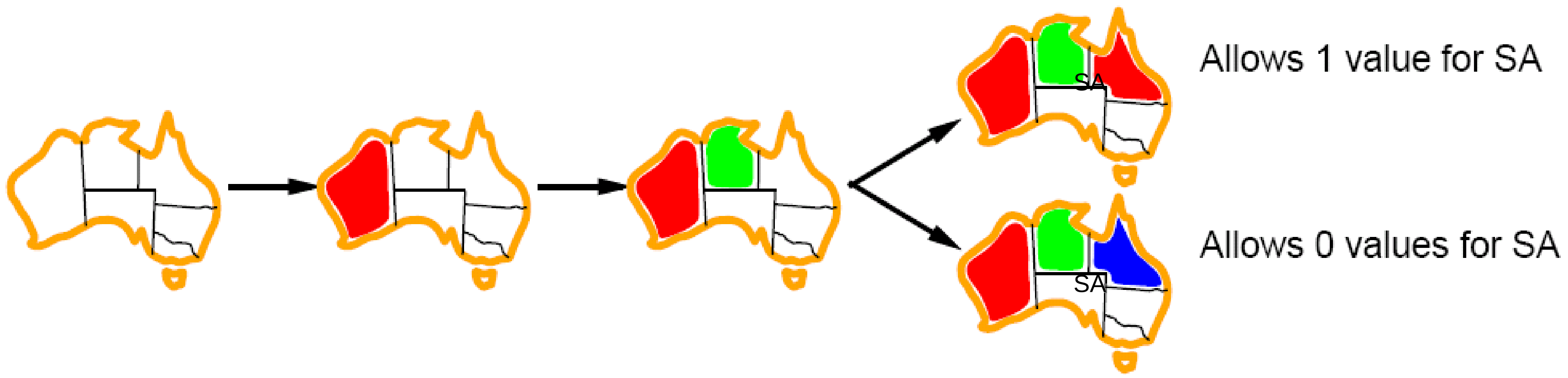
Which variable should be assigned next?



The Minimum Remaining Values heuristic is usually a more powerful guide but the Degree heuristic is very useful as a tie breaker.

Having selected a variable, in what order should its values be tried?

In what order should a variables values be tried?



Once a variable has been selected the algorithm must decide in which order it will examine its values.

Least constraining value heuristic: given a variable choose the least constraining value i.e. the one that leaves the maximum flexibility for subsequent variable assignments.

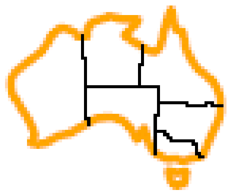
Of course if there is no solution to a problem or if we are trying to find all the solutions to a problem the order of the values does not matter.

Can we detect inevitable failure early?

So far our search algorithm considers the constraints on a variable only at the time that the variable is chosen for value assignment.

But by looking ahead at some of the constraints earlier in the search, or even before the search has started, we can drastically reduce the search space.

Can we detect inevitable failure early?



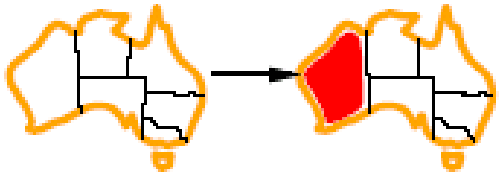
WA	NT	Q	NSW	V	SA	T
🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦

Forward checking idea: keep track of remaining legal values for unassigned variables.

Whenever a variable X is assigned, the forward checking process looks at each unassigned variable Y that is connected to X by a constraint and deletes from Y's domain any value that is inconsistent with the value chosen for X.

Terminate search when any variable has no legal values.

Can we detect inevitable failure early?



WA	NT	Q	NSW	V	SA	T
🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦
🟥	🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟩🟦	🟥🟩🟦

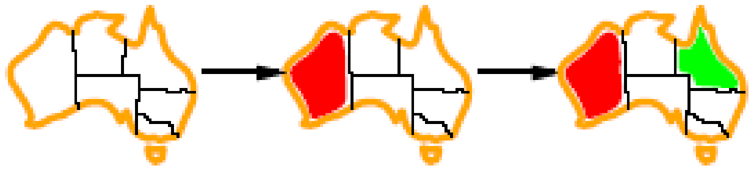
Forward checking example:

Assign *{WA=red}*

Effects on other variables connected by constraints with WA

- *NT can no longer be red*
- *SA can no longer be red*

Can we detect inevitable failure early?



WA	NT	Q	NSW	V	SA	T
🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦
🟥	🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟩🟦	🟥🟩🟦
🟥	🟦	🟩	🟥🟦	🟥🟩🟦	🟦	🟥🟩🟦

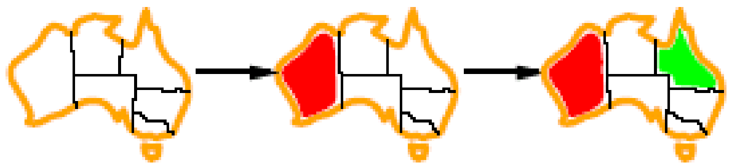
Forward checking example (cont.):

Assign *{Q=green}*

Effects on other variables connected by constraints with WA

- *NT can no longer be green*
- *NSW can no longer be green*
- *SA can no longer be green*

Can we detect inevitable failure early?



WA	NT	Q	NSW	V	SA	T
🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦
🟥	🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟩🟦	🟥🟩🟦
🟥	🟦	🟩	🟥🟦	🟥🟩🟦	🟦	🟥🟩🟦

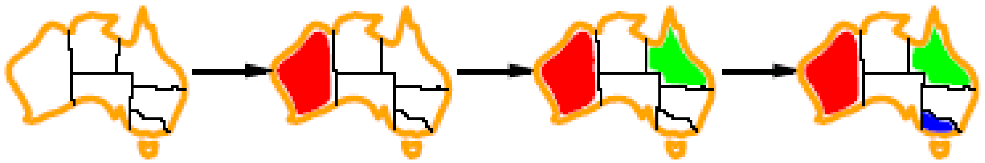
Forward checking example (cont):

Note: at this point the MRV heuristic will automatically select NT and SA next, why?

Because their domains are reduced to a single value.

We can view forward checking as an efficient way to incrementally compute the information that the MRV heuristic needs to do its job.

Can we detect inevitable failure early?



WA	NT	Q	NSW	V	SA	T
🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦
🟥	🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟩🟦	🟥🟩🟦
🟥	🟦	🟩	🟥🟦	🟥🟩🟦	🟦	🟥🟩🟦
🟥	🟦	🟩	🟥	🟦		🟥🟩🟦

Forward checking example (cont):

If *V* is assigned *blue*

Effects on other variables connected by constraints with WA

- *SA is empty*
- *NSW can no longer be blue*

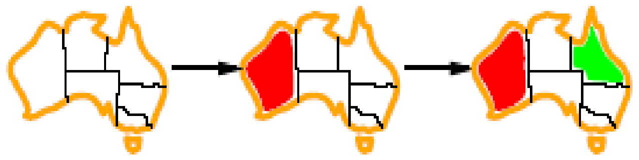
FC has detected that partial assignment is *inconsistent* with the constraints and backtracking can occur.

Can we detect inevitable failure early?

Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failure.

For example, during the map colouring forward checking example we reached a point where both NT and SA were forced to be blue.

But as they are adjacent they cannot have the same value! Forward checking did not detect this inconsis



WA	NT	Q	NSW	V	SA	T
🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦
🟥	🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟩🟦	🟥🟩🟦
🟥	🟦	🟩	🟥🟦	🟥🟩🟦	🟦	🟥🟩🟦

Can we detect inevitable failure early?

Constraint propagation is the general term for propagating the implications of a constraint on from one variable onto other variables:

In our map colouring example we need to propagate from WA and Q onto NT and SA, (as done by forward checking) and then onto the constraint between NT and SA to detect the inconsistency.

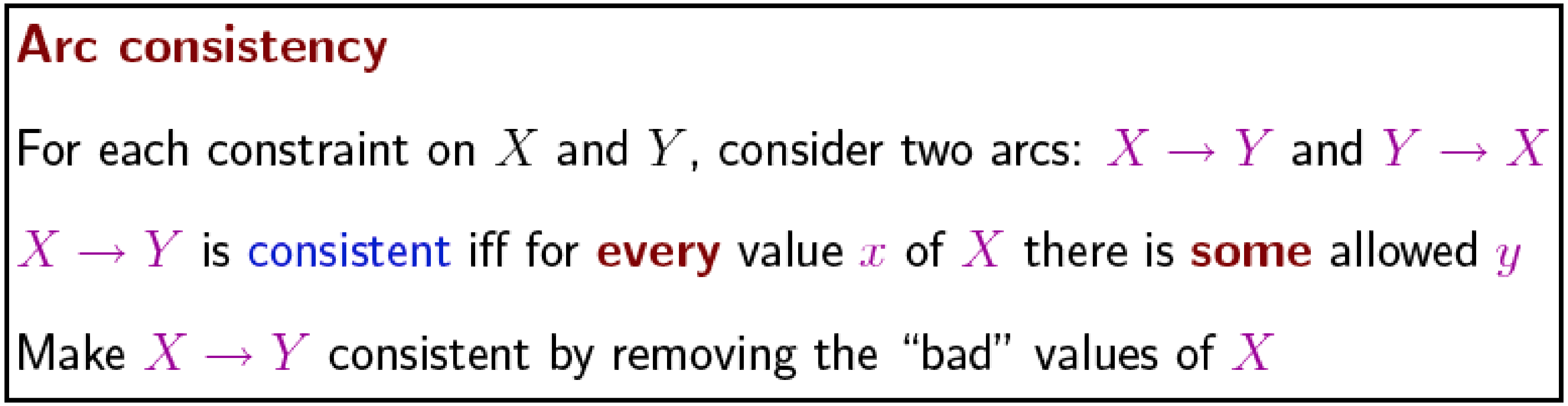
Moreover, we want to do this fast; it is no good reducing the amount of search if we spend more time propagating constraints than we would do searching.

Can we detect inevitable failure early?

Arc consistency:

Provides a fast method of constraint propagation that is substantially stronger than forward checking.

Here, "arc" refers to a directed arc in the constraint graph.



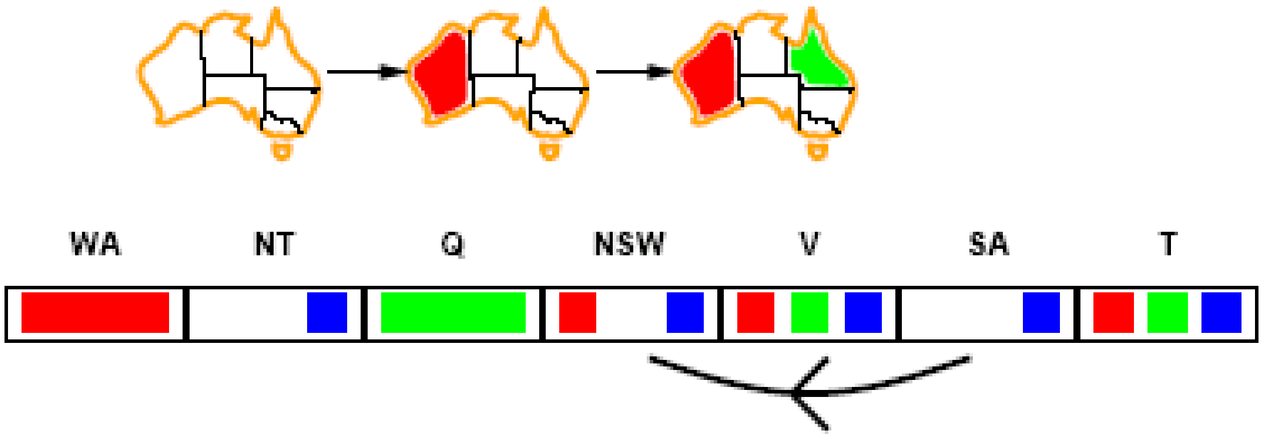
Arc consistency

For each constraint on X and Y, consider two arcs: $X \rightarrow Y$ and $Y \rightarrow X$

$X \rightarrow Y$ is consistent iff for **every** value x of X there is **some** allowed y

Make $X \rightarrow Y$ consistent by removing the "bad" values of X

Can we detect inevitable failure early?



Arc consistency:

X $\rightarrow$ *Y* is consistent iff

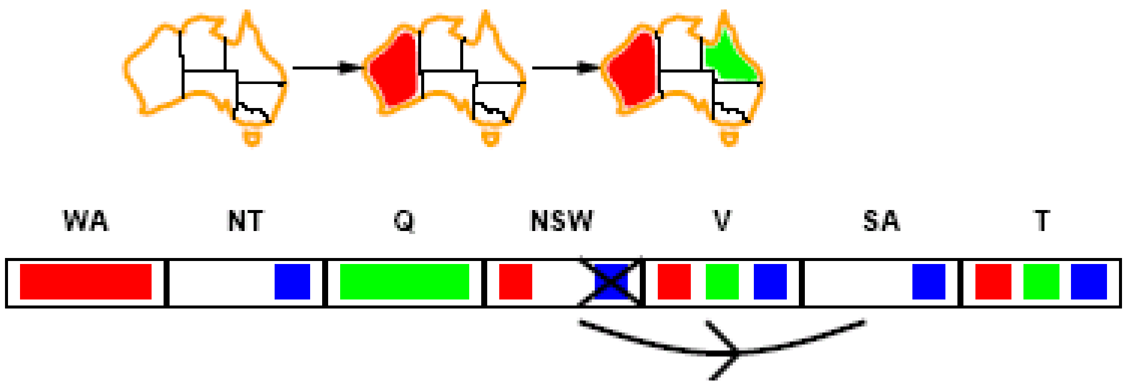
for *every* value *x* of *X* there is some allowed *y*

SA $\rightarrow$ *NSW* is consistent iff

SA=blue and *NSW=red*

Can we detect inevitable failure early?

Arc Consistency



X → *Y* is consistent iff

for *every* value x of *X* there is some allowed *y*

NSW → *SA* is consistent iff

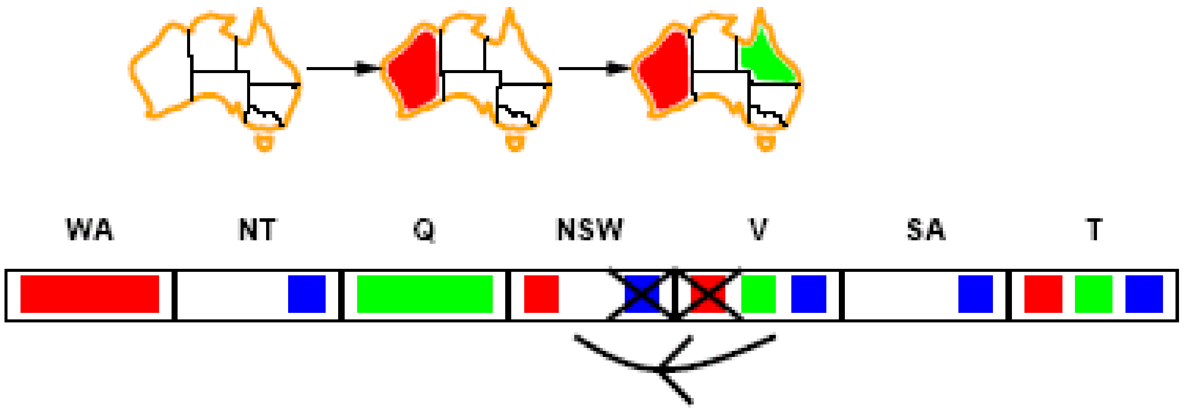
NSW=red and *SA=blue*

NSW=blue and SA=???

Arc *NSW* → *SA* can be made consistent by removing *blue* from *NSW*

Can we detect inevitable failure early?

Arc Consistency



Note that if a variable X loses a value, every arc W → X needs to be rechecked.

In this instance if NSW loose blue than the arc V → NSW must be rechecked.

V = blue and NSW=red
V = green and NSW = red
V = red and NSW ?

This results in red being removed from V

Can we detect inevitable failure early?

Arc consistency:

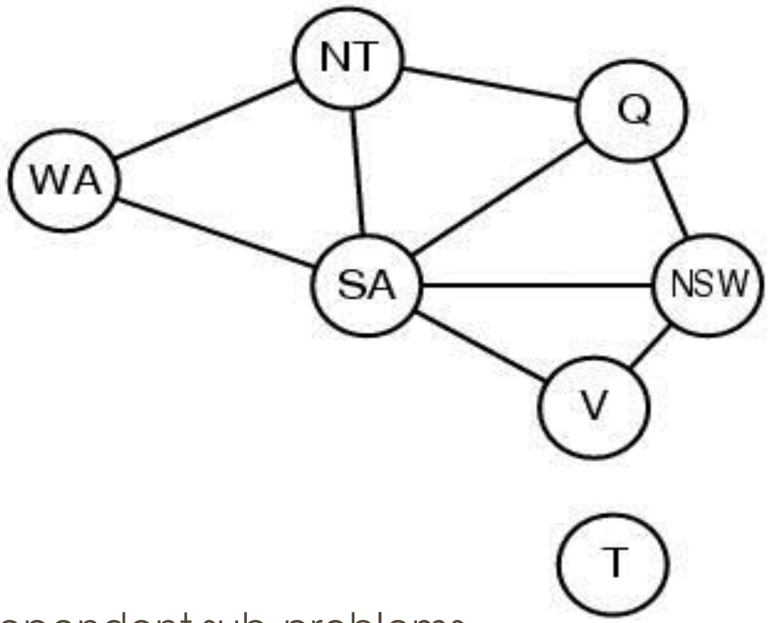
✓In general finds failures earlier and faster than forward-checking.

✓Finds all the failures that forward-checking would find, plus more.

X Doesn't find all failures - that's NP-hard.

How can the problem structure help to find a solution quickly?

Sometimes it is helpful to visualize a CSP as a constraint graph = nodes are variables, edges show constraints.



Sub-problem identification is important:

- Coloring Tasmania and mainland are independent sub-problems
- Identifiable as connected components of constrained graph. Each connected component corresponds to a sub-problem.

A solution for the overall CSP can be constructed by unifying the sub-problem solutions.

Improves performance

Summary

CSPs are a special kind of problem: states defined by values of a fixed set of variables, goal test defined by constraints on variable values

Variable ordering and value selection heuristics help significantly

Forward checking prevents assignments that lead to failure.

Constraint propagation does additional work to constrain values and detect inconsistencies.

The CSP representation allows analysis of problem structure.

Example: Crypt-arithmetic puzzle

```
    T W O
+
    T W O
-----------------
F O U R
```

Example: Crypt-arithmetic puzzle

```
   T W O
+
   T W O
-----------------
F O U R
```

Variables
F, O, R, T, U, W

Domains
{0, 1, 2, 3 ,4, 5, 6, 7, 8, 9 }

Constraints
$T \neq 0$, $F \neq 0$ (unary constraints)
$O + O = R + 10 \cdot X1$
$X1 + W + W = U + 10 \cdot X2$
$X2 + T + T = O + 10 \cdot X3$
$X3 = F$
ALL-DIFFERENT(F, O, R, T, U, W): $F \neq O$, $F \neq R$, etc.

Example: Crypt-arithmetic puzzle

```
   T W O
+
   T W O
-----------------
 F O U R
```

We can re-write the domains:

D_F , D_{X3} = {1}

D_{X2} , D_{X1} = {0, 1}

D_T = {1, 2, 3, 4, 5, 6, 7, 8, 9}

D_W , D_O, D_U , D_R = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

Example: Crypt-arithmetic puzzle

```
   T W O
+
   T W O
-----------------
 F O U R
```

(F, O, U, R, T, W, X_1, X_2, X_3)

(–, –, –, –, –, –, –, –, –)

(1, –, –, –, –, –, –, –, –)

Select F next via MRV

Assign F = 1. Remove 1 from all domains by forward checking.

Select X_3 next via MRV.

D_{X2} , D_{X1} = {0, 1}

D_T = {2, 3, 4, 5, 6, 7, 8, 9}

D_W , D_O, D_U , D_R = {0, 2, 3, 4, 5, 6, 7, 8, 9}

Example: Crypt-arithmetic puzzle

```
   T W O
+
   T W O
-----------------
 F O U R
```

$(F, O, U, R, T, W, X_1, X_2, X_3)$
$(1, -, -, -, -, -, -, -, 1)$

Assign $X_3 = 1$. R
Remove {2, 3, 4} from D_T by forward checking.
Select X_2 using MRV.

D_{X2} , D_{X1} = {0, 1}
D_T = {5, 6, 7, 8, 9}
D_W , D_O, D_U , D_R = {0, 2, 3, 4, 5, 6, 7, 8, 9}

Example: Crypt-arithmetic puzzle

```
   T W O
+
   T W O
-----------------
 F O U R
```

Etc…

Solution:

```
   8 3 6
+
   8 3 6
----------------
 1 6 7 2
```